Large Language Models in SAT Reasoning

HYNER Petr[1,2], DUŠEK František[3], ADAMCZYK David[1] and HŮLA Jan[1,3]

[1] *Institute for Research and Application of Fuzzy Modeling, University of Ostrava*
Ostrava, Czech Republic

[2] *Department of Informatics and Computers, University of Ostrava*
Ostrava, Czech Republic

[3] *Czech Technical University in Prague*
Prague, Czech Republic

E-mail: {petr.hyner, david.adamczyk}@osu.cz, koutefra@fit.cvut.cz, jan.hula@cvut.cz

1 Introduction

In this contribution, we describe the results of preliminary experiments aimed at testing whether an autoregressive Transformer can learn to imitate the steps of a sophisticated symbolic solver. Concretely, we train it to imitate the steps of a SAT solver based on *Conflict-Driven Clause Learning* (CDCL). We use our custom implementation of the solver, which contains only the most basic features.

The solver assigns Boolean values to variables in a CNF formula in a depth-first search manner (for some fixed order of variables) and backtracks whenever it finds that the current (partial) assignment is unsatisfiable. The process is implemented in a procedure we refer to as *Solve*. Inside the Solve procedure, the solver calls two other procedures: *Unit Propagation* and *Conflict Analysis*. Unit Propagation takes the current assignment and tries to deduce values for other unassigned variables. It can also detect that the current (partial) assignment is unsatisfiable (resulting in a conflict), in which case the Conflict Analysis procedure is invoked to analyze the assignment and determine the cause of the conflict. This procedure creates (learns) a new clause that prevents such a conflict from occurring again in later iterations of the solver. Adding such clauses to the set of original clauses of the CNF formula effectively restricts the search space.

We instrument the solver with logging code and record the solving traces for random CNF formulas of a certain size. The traces are divided among the three procedures in the solver (Solve, Unit Propagation, Conflict Analysis). This means that for each invocation of these procedures, we obtain one string describing the steps of the procedure, which serves as one training example for the model.

The model is trained on all traces recorded from training CNF formulas and then validated on traces from distinct validation CNF formulas.

Acknowledgement This contribution has been produced with the financial support of the European Union under the: Biography of Fake News with a Touch of AI: Dangerous Phenomenon through the Prism of Modern Human Sciences project no.: CZ.02.01.01/00/23_025/0008724 via the Operational Programme Jan Ámos Komenský. Model training and evaluation was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254).

Copyright © 2026 Authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

https://www.doi.org/10.15452/978-80-7599-515-5.2026.09

2 Experimental Setup

In this section, we will describe how we generate data, the chosen model architecture, and the training approach.

2.1 Data

We generate 1.2M synthetic execution traces from our CDCL solver implementation for SAT problems with 5-15 variables (in-distribution). The dataset is balanced across three trace types: 400K main solver loops, 400K unit propagation steps, and 400K conflict analysis procedures. Each trace represents a complete execution sequence of the corresponding algorithm component, including all intermediate state transitions and decisions.

The traces include oracle information blocks delimited by special "start reading" and "stop reading" tokens, which provide access to problem state during execution. These blocks contain current variable assignments, decision levels, the clause database, and reason clauses for implications. For example, an assignments read block might contain "x 1 = True, x 2 = False, x 3 = True", while clause read block provides the current formula in CNF form.

Additionally, we generate out-of-distribution test traces for problems with 16-25 variables using the same balanced composition. This OOD set tests the model's ability to generalize to larger problem instances not seen during training, which inherently require more complex reasoning to solve correctly.

We train four models: three task-specific models (one for each trace type) and one unified model trained on the mixed dataset containing all trace types. This allows us to evaluate both specialized performance as well as the model's ability to learn all three tasks at the same time.

The in-distribution and out-of-distribution datasets show substantial differences in complexity. We measured the number of solver method invocations per problem and find that for IID instances, the mean number of calls is 11.5 for Unit Propagation, 4.1 for Conflict Analysis, and 8.2 for Solve, while OOD instances show approximately 2× growth across all trace types: Unit Propagation averages 24.9 calls, Conflict Analysis 9.2 calls, and Solve 18.3 calls. This increased computational complexity is reflected in trace sequence lengths, which grow from a mean of 2,333 tokens in IID data to 3,858 tokens in OOD data.

2.2 Model

We use small-scale GPT-NeoX models [1] with 12 transformer layers, 8 attention heads, and embedding dimension of 256, resulting in approximately 9.5M parameters. We set a context window of 4096 tokens and use Rotary Position Embeddings (RoPE) [3].

Our vocabulary consists of only 88 domain-specific tokens representing CDCL algorithm operations and SAT formula elements. These include control flow tokens (`SOLVE_BEGIN`, `UNIT_PROPAGATION_BEGIN`), state markers (`READ_ASSIGNMENTS`, `READ_CLAUSES`), logical operations (`BRANCHING_VARIABLE`, `PROPAGATED`, `BACKTRACK`), and SAT-specific symbols (variable identifiers, literals, clause markers). We use a custom WordLevel tokenizer with whitespace splitting, as we specifically structure the data so that subword tokenization becomes unnecessary.

During training, we apply causal masking as in standard autoregressive models, where each token attends only to preceding tokens. Additionally, we set the loss weight to zero for two groups of tokens: (1) the initial `[BOS]` token and the following task identifier (`SOLVE_BEGIN`, `UNIT_PROPAGATION_BEGIN`, or `ANALYZE_CONFLICT_BEGIN`), and (2) all tokens within `READ_BEGIN` ... `READ_END` blocks. These oracle-provided segments (e.g., variable assignments or clause

lists) remain visible to the model through causal attention but are excluded from gradient computation. Consequently, the model can condition its next actions on oracle content without being trained to reproduce it, establishing a read-only scratchpad interaction consistent with inference-time behavior (see Figure 1).

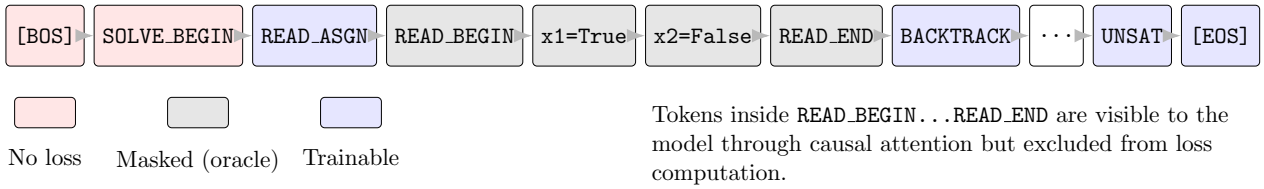

Figure 1: Training-time masking. Tokens within `READ_BEGIN...READ_END` are visible but excluded from loss.

3 Interactive Scratchpad

To enable structured reasoning over solver states, we introduce an *interactive scratchpad*—a tokenized key–value memory that the model reads and writes through explicit commands. Each key (e.g., `assignments`, `decision_levels`) stores a sequence of token IDs representing part of the CDCL state. When the model emits a `READ_<key>` command, the environment injects the corresponding content enclosed in `READ_BEGIN...READ_END`, making it visible through causal attention but excluded from loss computation (see Figure 1). Conversely, `WRITE_<key>` commands update or append to the stored values.

The `CDCLScratchpad` extends this interface with operations such as `level_up` and `backtrack`, which modify symbolic structures like decision levels and reason clauses by detokenizing, editing, and re-tokenizing the relevant fields.

During inference, the `AutoregressiveCDCLEnvironment` orchestrates the interaction between the model and the scratchpad. It interprets emitted tokens, injects oracle reads, stores writes, and temporarily switches contexts for subprocedures (e.g., Unit Propagation, Conflict Analysis). This setup transforms the language model into a state-aware reasoner that dynamically interacts with its symbolic environment rather than passively replaying static traces, following the broader idea of scratchpad-based reasoning introduced by Nye et al. [2].

4 Preliminary Results

We evaluate model generalization and performance using two metrics. **Token-level accuracy** measures the proportion of correctly predicted tokens at corresponding positions:

$$\mathrm{Acc}_{\mathrm{token}} = \frac{1}{N}\sum_{i=1}^{N}\frac{|\{j : p_i[j] = g_i[j]\}|}{\max(|p_i|, |g_i|)}$$

where p_i and g_i are the predicted and ground truth token sequences for example i, and N is the number of test examples. This formulation penalizes both missing and extraneous tokens.

Exact match accuracy requires complete equality:

$$\mathrm{Acc}_{\mathrm{exact}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}[p_i = g_i]$$

Both metrics assess syntactic similarity to reference CDCL traces rather than semantic validity of the solutions. Both are computed on in-distribution (IID, 5-15 variables) and out-of-distribution (OOD, 16-25 variables) test sets. Tables 1 and 2 show the results.

Table 1: Token-level accuracy on CDCL trace generation tasks (n=512). Task-specific models are evaluated only on their corresponding task.

	Solve		**Unit Propagation**		**Conflict Analysis**	
Model	IID	OOD	IID	OOD	IID	OOD
Solve-only	0.9275	0.8740	—	—	—	—
UP-only	—	—	0.9985	0.9767	—	—
AC-only	—	—	—	—	0.9938	0.5745
Mixed	0.9999	0.9711	0.9982	0.8672	0.9962	0.7315

Table 2: Exact match accuracy on CDCL trace generation tasks (n=512). Task-specific models are evaluated only on their corresponding task.

	Solve		**Unit Propagation**		**Conflict Analysis**	
Model	IID	OOD	IID	OOD	IID	OOD
Solve-only	0.8438	0.6406	—	—	—	—
UP-only	—	—	0.9629	0.6973	—	—
AC-only	—	—	—	—	0.9531	0.4043
Mixed	0.9961	0.8770	0.9688	0.6504	0.9434	0.4707

5 Conclusion

We demonstrated that small autoregressive transformers can learn to imitate CDCL SAT solver operations with high accuracy on in-distribution problems (5-15 variables), achieving 92-99% token-level accuracy using an interactive scratchpad mechanism. While the models capture common algorithmic patterns, they struggle with the increased complexity of out-of-distribution instances with more variables and longer solving traces, suggesting that larger model capacity or improved training strategies are needed for robust generalization.

References

[1] Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Th´erien, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023.

[2] Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Witold Michalewski, Jacob Austin, David Bieber, David Martin Dohan, Aitor Lewkowycz, Maarten Paul Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. https://arxiv.org/abs/2112.00114.

[3] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.